Retrieval Augmented Generation (RAGs)

Retrieval Augmented Generation (RAGs)

What we will cover;

1. RAG Introduction:

We'll start with what RAG actually is - how it lets AI systems search through your documents and give accurate, grounded answers instead of making things up.

2. Ingestion Pipeline

You'll learn how to take raw documents and convert them into a searchable format - loading files, splitting text, and storing everything in a vector database.

3. Retrieval Pipeline

Then we'll cover how to search through that data - turning questions into vectors and finding the most relevant document chunks

4. Cosine Similarity

We'll dive into the math behind similarity scoring - how the system actually decides which document chunks are relevant to your question.

Retrieval Augmented Generation (RAGs)

What we will cover (contd.) ;

5. One-off RAG:

First, we'll build a simple RAG system that answers individual questions without any memory.

6. Conversational RAG

Then we'll upgrade it to handle conversations - where follow-up questions actually work properly.

7. Importance of Chunking:

Here's where things get interesting - and where most tutorials fall short. We'll dive deep into chunking strategies because how you split your text makes or breaks your RAG system.

8. RecursiveCharacterTextSplitter

You'll master the most popular chunking method that tries multiple strategies to split text intelligently.

Retrieval Augmented Generation (RAGs)

What we will cover (contd.) ;

9. Document-Specific Splitting

Learn how to handle different file types - PDFs, Word docs, and markdown files each need different approaches.

10. Semantic Splitting

We'll cover the cutting-edge approach that actually understands content meaning to decide where to split text

11. Agentic Splitting

Finally, we'll look at using AI itself to make smart decisions about how to chunk your documents.

12. Types of Retrieval Methods

Beyond basic similarity search, you'll learn advanced techniques like similarity threshold and max marginal relevance.

13. Multi-Query Retrieval

See how rephrasing the same question in multiple ways helps retrieve chunks that use different terminology or phrasing than your original query.

Retrieval Augmented Generation (RAGs)

What we will cover (contd.) ;

14. Reciprocal Rank Fusion (RRF):

Learn how to intelligently combine and score results from multiple query variations using reciprocal rank fusion

15. BM25 Retrieval

Understand the algorithm behind keyword search and why it's still relevant in the age of vector embeddings and RAGs.

16. Hybrid Search

Combine the best of both worlds - vector search for meaning and traditional keyword search for precision, using RRF to merge the results.

17. Reranking:

Top it all off with reranking techniques to further improve your search quality.

By the end of this series, you'll have everything you need to build production-ready RAG systems. Let's dive in!

Retrieval Augmented Generation (RAGs)

Definition:

RAGs is a method where we combine LLMs with a retrieval system.

This retrieval system can search through vast sources of external information - like documents, databases, or knowledge bases whenever the LLM needs additional knowledge to give you better answers

At the same time, it also makes sure the LLM is not overloaded with bigger prompts

Retrieval Augmented Generation (RAGs)

Real-world example:

Let's say you're working at a company with 100s of internal documents (like policy guidelines, technical specs, customer support documents, etc)

Now, if you have a question that could be answered by one of these documents, it would be unrealistic to dump all 100 files into the LLM and ask that question

It is unrealistic because LLMs have a "context window", meaning, a limited number of information it can process at a given time.

But with LLM + RAGs, this process can be streamlined.

So the next time when you have a question, you can just ask the RAG system in simple English and it retrieves the most relevant info from those documents and uses it to give you an accurate answer

AI Model Context Windows vs Company Document Sizes

Showing the massive gap between what AI models can process and real enterprise data volumes

🤖 Latest AI Models (2025)

⚠️ ENTERPRISE SCALE ⚠️

🏢 Company Document Repositories

Reality Check: Enterprise data centers (1 PB) contain ~13,000x more data than the largest AI model can process. Even a small company's 2GB is ~26x larger than GPT-5's context window. Companies need RAG systems and smart chunking to work with AI effectively.

Retrieval Augmented Generation (RAGs)

What are tokens?

In the context of language models, a token is a unit of text that the LLM model processes.

Tokens can be as short as one character or as long as one word, depending on the language and structure of the text

For example, the word "hello" is one token, while the phrase "I'm" is typically broken down into two tokens: "I" and "'m."

Retrieval Augmented Generation (RAGs)

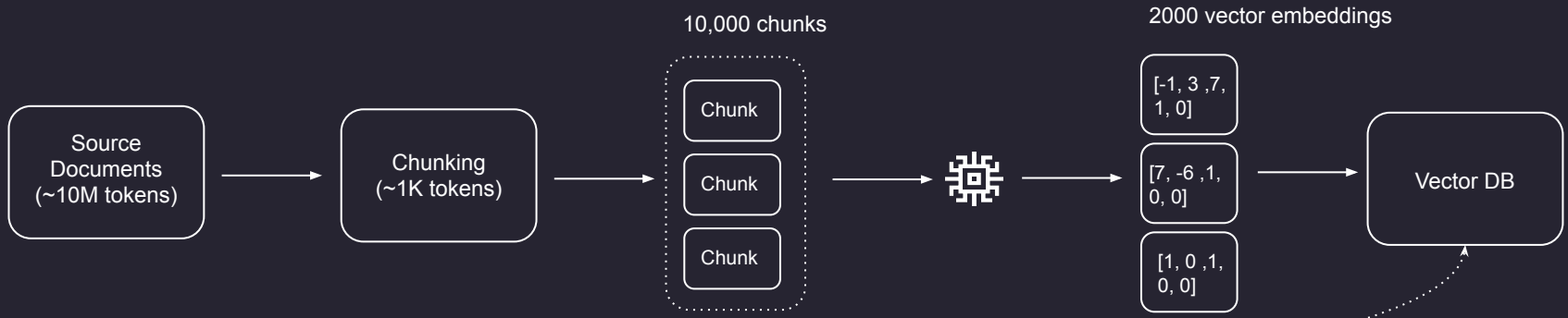
What are tokens? (contd.)

Understanding tokens are crucial because LLMs have a limit on how many tokens they can handle at once, referred to as the "context window."

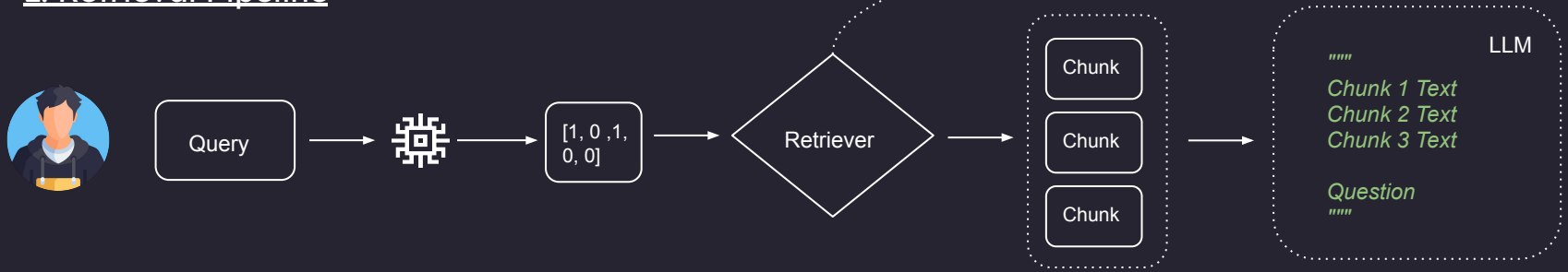
This means that if we have a PDF that's 10 million tokens long, we can't feed it to the model in one go.

Retrieval Augmented Generation (RAG) System

1. Knowledge-base construction (Ingestion pipeline):



2. Retrieval Pipeline



Embeddings and Vector DBs

What are embeddings?

A vector embedding is a mathematical representation of words, sentences or even images.

For example, the word "cat" can have a vector embedding that can look something like this: [34, 21, 7.5]

So this is the computer's way of making sense of the word "cat"

Basically each of these numbers can represent a certain aspect of the word "cat", like "small", "furry", etc.

Embeddings and Vector DBs

Examples:

Cat - [34, 8, 7.5]

Kitten - [33, 8, 2]

Elephant - [2, 62, 2]

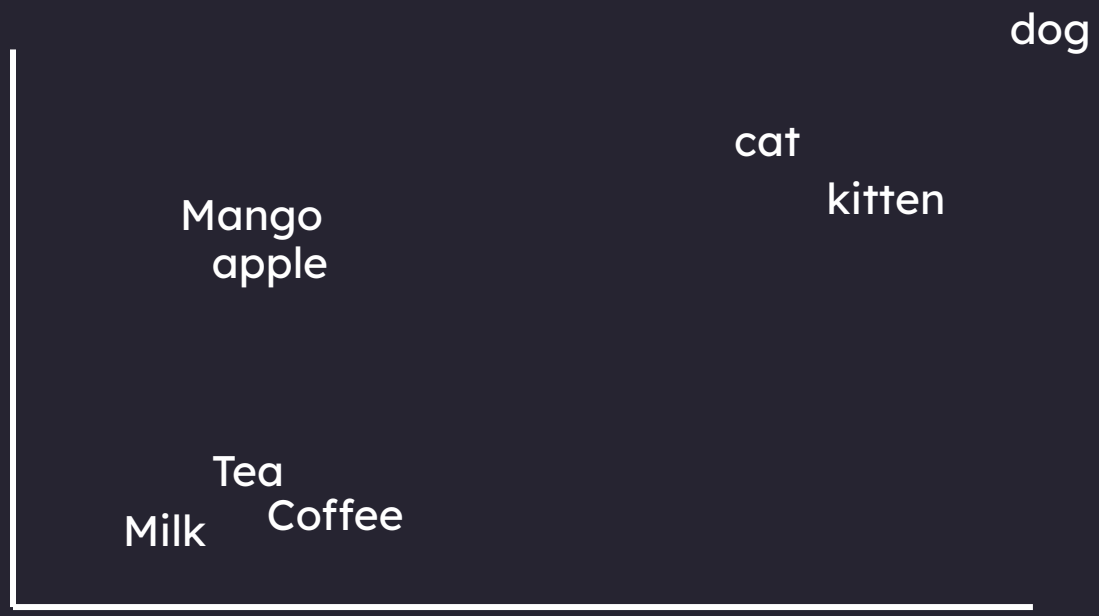
Dog - [47, 8, 2]

Each number in the vector embedding is called a dimension.

Words with similar semantic meaning tend to have dimensions that are closer to each other.

In other words, when a word's vector embedding is mapped in a multi-dimensional space, the spot where "cat" exists will be closer to where the word "kitten" exists

Embeddings and Vector DBs



Embeddings and Vector DBs

Examples:

Cat - [34, 8, 7.5]

Kitten - [33, 8, 2]

Elephant - [2, 62, 2]

Dog - [22, 8, 2]

Even though we only see 3 dimensions in each vector embedding, in reality, popular embedding models like OpenAI's "text-embedding-3-large" transform text to upto 3,072 dimensions in each vector embedding

Note that if we embed a small word like "cat" or a large paragraph, we always get one vector embedding output that has 3072 dimensions.

Advantage of using higher dimensional embedding models is, it captures more semantic information about the text passage. Downside is, it is slightly more expensive to embed as well as store.

Popular Embedding Models

There are many embedding model providers out there, but let's focus on the most popular ones you'll encounter:

OpenAI (Most Popular Choice):

- text-embedding-3-small: 1536 dimensions (default) - Great for most use cases (Can reduce to: 512, 1024, or any size ≤1536)
- text-embedding-3-large: 3072 dimensions (default) - Best performance, but costs more (Can reduce to: 256, 512, 1024, 1536, or any size ≤3072)
- Both models let you reduce dimensions to save storage costs without losing much quality

Other Notable Players:

- Cohere: Strong multilingual support, good for international applications
- Voyage AI: Often performs well in benchmarks
- Mistral: Open-source option that's been showing impressive results

Embeddings and Vector DBs

Vector DB definition:

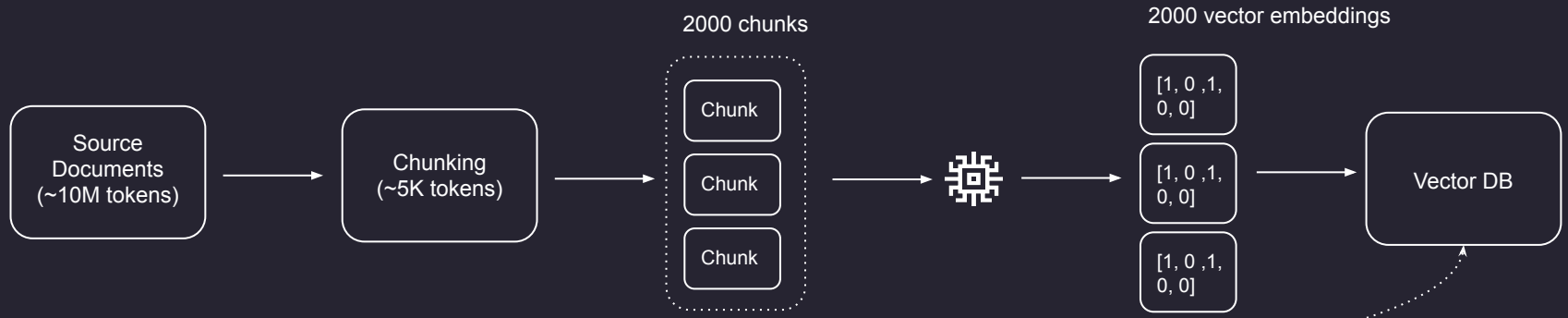
A database built to store all the vector embeddings is considered a vector database.

To store the vectors, we can use:

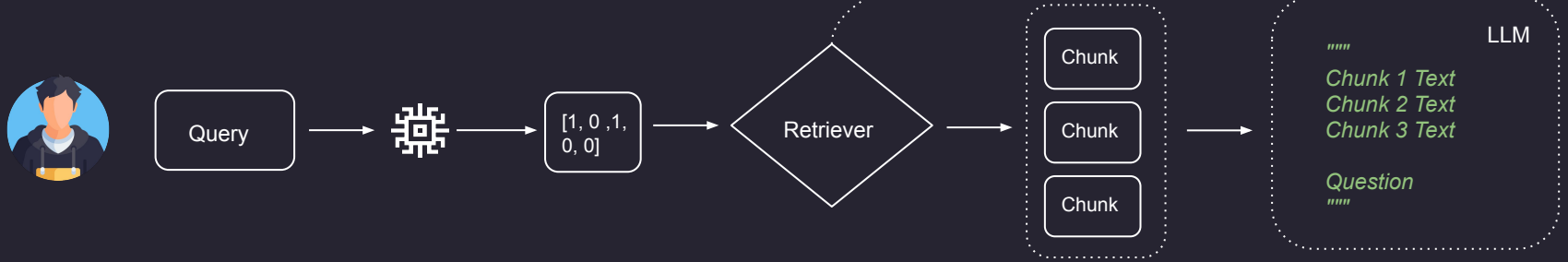
- Specialised Vector Databases - Pinecone, Weaviate, ChromaDB, FAISS
- Regular SQL Database - offers structured data storage and retrieval

Retrieval Augmented Generation (RAG) System

1. Knowledge-base construction (Ingestion pipeline):



2. Retrieval Pipeline



Embeddings and Vector DBs

Consistency is key:

You MUST use the same embedding model for both your documents AND your user queries.

Think of embeddings like different languages. If you embed your documents using text-embedding-3-small but embed user queries using text-embedding-3-large, they literally can't understand each other

Even within the same model, you must use the exact same dimensions everywhere.

For example: If you embed your documents using text-embedding-3-large with 1536 dimensions, you MUST also embed user queries with text-embedding-3-large at exactly 1536 dimensions.

Pick one embedding model at the start and stick with it throughout your entire RAG pipeline. If you want to switch later, you'll need to re-embed ALL your documents.

Cosine Similarity

Cosine Similarity is the reason why we're able to fetch the matching chunks for a user query from the vector database

So how does it work?

Definition: Cosine similarity measures the angle between vectors, not their magnitude

The cosine similarity values range from 0 to 1, with 0 being the least similar and 1 being the most similar

Formula:

cosine_similarity = (A · B) / (||A|| × ||B||)

History-aware generation

The Key Difference:

In **basic RAG**, each query is treated independently. The retriever takes your exact question and searches for chunks.

In **history-aware RAG**, there's one crucial extra step: **query reformulation.**

Before searching, the system looks at the conversation history and rewrites vague or context-dependent questions into clear, standalone questions.

Why This Matters: Follow-up Questions

Humans naturally ask follow-up questions using pronouns, references, and assumptions based on previous conversation. These questions are often unsearchable on their own.

History-aware generation

Examples with Company Documents

Conversation 1:

- **User:** "Tell me about NVIDIA's latest GPU architecture"
- **AI:** "NVIDIA's latest architecture is Hopper, featuring H100 chips..."
- **User:** "What's their revenue from it?"

Without History-Aware Retrieval:

Searches for: "What's their revenue from it?"

Vector embedding searches for words like "their", "revenue", "it"

Result: Poor/no results (embedding doesn't know what "their" or "it" refers to)

With History-Aware Retrieval:

Reformulates to: "What's NVIDIA's revenue from Hopper GPU architecture?"

Result: Finds relevant financial documents

RAG-chunking Strategies

RAG Ingestion Pipeline:

- Document Loading (PDF, DOCX, TXT → text)
- Text Chunking (long text → smaller pieces)
- Embedding (text chunks → vectors)
- Storage (vectors → vector database)

Chunking is the **critical** second step - it determines how your content gets divided for retrieval.

Your RAG system doesn't search entire documents. It searches chunks. So the final answer generation quality depends on those chunks.

Bad chunking breaks everything downstream - even perfect embeddings can't fix poorly split content.

RAG-chunking Strategies

The Problem with Basic Chunking

In our first videos, we used CharacterTextSplitter - it just cuts text at fixed character counts. Simple, but crude.

Example with Tesla document:

Chunk 1: "Tesla's Q3 revenue was $25.2B, up from $21.3B in Q2. The increase was driven by record Model Y sales which reached 350,000 units. However, production costs rose by 12% due to supply chain..."

Chunk 2: "...challenges and inflation. Elon Musk stated that the company expects to maintain growth through 2024 despite economic headwinds. The Cybertruck launch has been delayed again..."

Problems:

1. Splits mid-sentence ("supply chain..." / "...challenges")

2. Breaks related concepts apart

3. Context gets lost across chunks

4. Poor retrieval quality

RAG-chunking Strategies

Why Better Chunking Matters:

Your retrieval and the final answer generation is only as good as your chunks. Bad chunks = bad answers.

1. **Too small:** Lacks context
2. **Too large:** Too much noise, hits embedding model/context window limits
3. **Poor boundaries:** Splits related information
4. **No structure awareness:** Ignores document format

RAG-chunking Strategies

The 5 Chunking Strategies We'll Cover

1. CharacterTextSplitter (Beyond basic chunk_size)

- Custom separators (split on specific patterns)
- Still useful for simple, uniform documents or when speed matters most

2. RecursiveCharacterTextSplitter (Upgrade from CharacterTextSplitter)

- Tries to split at natural boundaries (paragraphs, sentences, words)
- Falls back gracefully if chunks too big
- Preserves more context than basic splitting

RAG-chunking Strategies

The 5 Chunking Strategies We'll Cover

3. Document-Specific Splitting (Respects document structure)

- PDF: Splits by pages, sections, headers
- Markdown: Splits by headers, code blocks, lists
- Each document type gets appropriate treatment

4. Semantic Splitting (Content-aware boundaries)

- Uses embeddings to detect topic shifts
- Keeps related concepts together
- Splits when meaning changes, not just by size
- More intelligent but computationally expensive

RAG-chunking Strategies

The 5 Chunking Strategies We'll Cover

5. Agentic Splitting (AI-powered chunking)

- LLM analyzes content and decides optimal splits
- Can understand complex relationships
- Adapts to content type automatically
- Most sophisticated but slowest/most expensive

CharacterTextSplitter

How it works:

CharacterTextSplitter doesn't just cut at character limits. It follows a split-first, merge-second approach.

1. **Split:** Break text at separators (default: \n\n)
2. **Merge:** Combine pieces until hitting chunk_size limit

Example:

```
"Tesla's Q3 Results

Tesla reported record revenue of $25.2B in Q3 2024.

Model Y Performance

The Model Y became the best-selling vehicle globally, with 350,000 units sold.

Production Challenges

Supply chain issues caused a 12% increase in production costs."
```

CharacterTextSplitter Configuration:

chunk_size: 100 characters
separator: \n\n (default)

CharacterTextSplitter

How it works:

CharacterTextSplitter doesn't just cut at character limits. It follows a split-first, merge-second approach.

1. **Split:** Break text at separators (default: \n\n)
2. **Merge:** Combine pieces until hitting chunk_size limit

Example:

```
"Tesla's Q3 Results

Tesla reported record revenue of $25.2B in Q3 2024.

Model Y Performance

The Model Y became the best-selling vehicle globally, with 350,000 units sold.

Production Challenges

Supply chain issues caused a 12% increase in production costs."
```

CharacterTextSplitter Configuration:

chunk_size: 100 characters
separator: \n\n (default)

CharacterTextSplitter

Step 1: Split at \n\n:

```
Piece 1: "Tesla's Q3 Results" (18 chars)
Piece 2: "Tesla reported record revenue of $25.2B in Q3 2024." (51 chars)
Piece 3: "Model Y Performance" (19 chars)
Piece 4: "The Model Y became the best-selling vehicle globally, with 350,000 units sold." (78 chars)
Piece 5: "Production Challenges" (21 chars)
Piece 6: "Supply chain issues caused a 12% increase in production costs." (62 chars)
```

CharacterTextSplitter

Step 1: Split at \n\n:

```
Piece 1: "Tesla's Q3 Results" (18 chars)
Piece 2: "Tesla reported record revenue of $25.2B in Q3 2024." (51 chars)
Piece 3: "Model Y Performance" (19 chars)
Piece 4: "The Model Y became the best-selling vehicle globally, with 350,000 units sold." (78 chars)
Piece 5: "Production Challenges" (21 chars)
Piece 6: "Supply chain issues caused a 12% increase in production costs." (62 chars)
```

CharacterTextSplitter

Step 2 - Merge until chunk_size (let's say 100 chars):

```
Final Chunk 1: "Tesla's Q3 Results\n\nTesla reported record revenue of $25.2B in Q3 2024.\n\nModel Y Performance" (92 chars)

Chunk 2: "The Model Y became the best-selling vehicle globally, with 350,000 units sold." (78 chars)

Chunk 3: "Production Challenges\n\nSupply chain issues caused a 12% increase in production costs." (85 chars)
```

CharacterTextSplitter

The Problem: When a single splits exceeds chunk_size

What happens if a single piece is larger than chunk_size of 100?

```
text = "This sentence is written to be deliberately long, flowing, and continuous, without breaking into
shorter sections. This nearly reaches two hundred characters in total length overall. "
```

In the above text, the number of characters is close to 200 characters.

But the CharacterTextSplitter cannot split this text further because there's no "double new line"

Result: CharacterTextSplitter keeps it as-is, even though the chunk exceeds the limit

SemanticChunker

How it works:

Semantic chunking breaks up long documents into meaningful pieces by finding where topics naturally change.

Instead of cutting at random word counts, it uses AI embeddings to understand the semantic meaning of sentences.

If one sentence talks about a certain topic and the next sentence talks about an entire different topic, then it means we need to chunk

3-Step Process:

1. Encode: Convert each sentence into embeddings (numerical vectors)

2. Compare: Calculate similarity score between nearby sentences

3. Split: Create boundaries where similarity drops **significantly**

SemanticChunker

How it works:

Example with content:

```
Sentences:
1. "Tesla reported record revenue of $25.2B in Q3 2024."
2. "The company exceeded analyst expectations by 15%."
3. "Revenue growth was driven by strong vehicle deliveries."
4. "The Model Y became the best-selling vehicle globally."
5. "Customer satisfaction ratings reached an all-time high."
```

SemanticChunker

How it works:

Step 1 - Create embeddings:

Each sentence is converted into a vector embedding

Step 2 - Calculate similarity scores:

```
Similarity between sentences 1-2: 0.85 (both about financial results)
Similarity between sentences 2-3: 0.78 (both about revenue performance)
Similarity between sentences 3-4: 0.42 (topic shift: revenue → vehicle sales)
Similarity between sentences 4-5: 0.71 (both about Model Y performance)
```

Step 3 - Detect breakpoints:

The most common breakpoint criteria to decide semantic splitting with SemanticChunker is with "percentiles"

SemanticChunker

What is a percentile?

A percentile tells you what percentage of values are below a certain number.

If you're at the 70th percentile, it means 70% of all values are below yours, and only 30% are above yours.

Example: CAT or SAT Scores

If 200,000 students took a CAT or SAT exam and you scored at the 95th percentile, it means:

- You scored better than 190,000 students (95% of all test-takers)
- Only 10,000 students (5%) scored higher than you
- You're in the top 5% of all test-takers

Percentiles are all about relative ranking, not absolute scores. The same score can be great or poor depending on how others performed.

For example, scoring 120/200 might sound average, but if it's the 85th percentile, you actually beat 85% of everyone who took the exam!

SemanticChunker

How do we calculate the percentile?

Step 1: Put all scores in order from lowest to highest

Example: [45, 62, 78, 85, 92, 105, 118, 135, 147, 168]

Step 2: Pick your percentile (let's say 70th percentile)

Step 3: Calculate which position to look at

70% of 10 scores = 0.70 × 10 = 7th position

Step 4: Find the score at that position

The 7th score in our list is 118

70th percentile = 118

This means 70% of all scores (7 out of 10) are below 118

SemanticChunker

How it works:

Step 1 - Create embeddings:

Each sentence is converted into a vector embedding

Step 2 - Calculate similarity scores:

```
Similarity between sentences 1-2: 0.85 (both about financial results)
Similarity between sentences 2-3: 0.78 (both about revenue performance)
Similarity between sentences 3-4: 0.42 (topic shift: revenue → vehicle sales)
Similarity between sentences 4-5: 0.71 (both about Model Y performance)
```

Step 3 - Detect breakpoints:

The most common breakpoint criteria to decide semantic splitting with SemanticChunker is with "percentiles"

SemanticChunker

How This Translates to Semantic Chunking

Instead of exam scores, we now have similarity scores between sentences:

Example: Tesla document with similarity scores between consecutive sentences: [0.85, 0.78, 0.42, 0.71, 0.95]

Step 1: Put similarity scores in order from lowest to highest
[0.42, 0.71, 0.78, 0.85, 0.95]

Step 2: Pick your percentile (40th percentile)

Step 3: Calculate which position to look at 40% of 5 scores = 0.40 × 5 = 2nd position

Step 4: Find the score at that position

The 2nd score in our list is 0.71

40th percentile = 0.71

SemanticChunker

Apply the Split Rule

Split wherever similarity ≤ 0.71

Looking at our original sequence:

0.85 → keep together (above 0.71)

0.78 → keep together (above 0.71)

0.42 → SPLIT HERE (below 0.71)

0.71 → keep together (equal to 0.71)

0.95 → keep together (above 0.71)

Result: Split after sentence 3 (the topic shift from revenue to vehicle sales)

SemanticChunker

Why 70th Percentile?

Common range: 60th-90th percentile works well for most documents

70th percentile is popular because:

- Not too aggressive (wouldn't split at every small dip)
- Not too conservative (wouldn't miss obvious topic changes)

SemanticChunker

Why Use Percentile?

Problem: Different documents have different similarity patterns

Example 1 - Academic Paper: Most similarities are high [0.85, 0.88, 0.91, 0.87, 0.89]

Example 2 - News Article: Most similarities are lower [0.45, 0.52, 0.38, 0.61, 0.43]

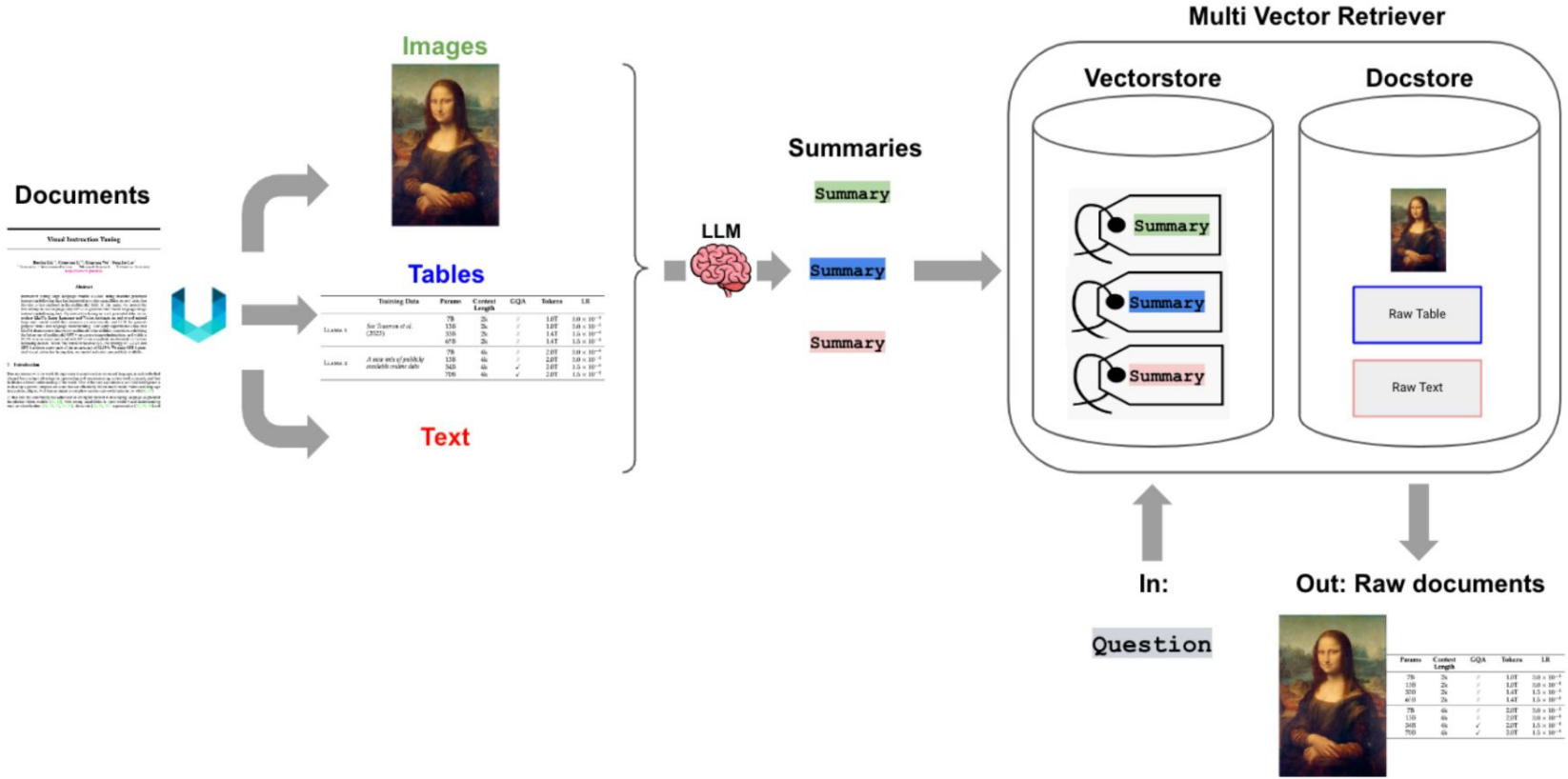
If we used a fixed threshold like 0.70:

- Academic paper: Would NEVER split (all scores above 0.70)
- News article: Would split EVERYWHERE (all scores below 0.70)

Percentile solves this!

70th percentile automatically adapts:

Academic paper: 70th percentile = 0.88 (finds the relatively weakest connections)
News article: 70th percentile = 0.52 (finds the relatively weakest connections)


Multi Vector Retriever
Images
Documents
Tables
Text
LLM
Summaries
Summary
Summary
Summary
Vectorstore
Docstore
Summary
Summary
Summary
Raw Table
Raw Text
In:
Question
Out: Raw documents

Multi-Modal Document Retrieval With Unstructured.io

Dependencies:

For Linux:

apt-get install poppler-utils tesseract-ocr libmagic-dev

For Mac:

brew install poppler tesseract libmagic

1. **Poppler** - Enables Unstructured to parse PDF files by extracting text, handling different PDF structures, and dealing with complex layouts.

2. **Tesseract** - Provides OCR capabilities so Unstructured can extract text from scanned documents, images, and image-based PDFs.

3. **libmagic** - Allows Unstructured to automatically identify file types based on content signatures rather than just file extensions.

Maximum Marginal Relevance (MMR)

The Problem MMR Solves:

Imagine you ask: "Tell me about Python programming"

Regular similarity search might return:

1. "Python is a programming language..."
2. "Python is a popular programming language..."
3. "Python programming language is widely used..."

The problem with this approach is all three documents are basically saying the same thing. You're getting redundant repeated information.

Maximum Marginal Relevance (MMR)

How MMR Works:

MMR uses a two-step process:

1. **First:** Find chunks relevant to your query (like normal similarity)

2. **Then:** Among those relevant chunks, pick the most diverse ones

It's like having a smart assistant that says: "I found 20 articles about Python, but instead of giving you the 3 most similar ones, let me give you 3 different aspects - one about syntax, one about applications, and one about libraries."

Formula MMR Follows:

MMR balances two competing goals:

- Relevance: How well does this chunk answer the query?

- Diversity: How different is this chunk from what I've already selected?

Maximum Marginal Relevance (MMR)

When to use MMR:

Use MMR when:

1. Your documents might have overlapping content
2. You want a "well-rounded" answer covering different aspects
3. You're doing research and want diverse perspectives

Don't use MMR when:

1. You want the absolute most relevant results
2. Your chunks are already quite diverse
3. Speed is critical (MMR is slower)

Reciprocal Rank Fusion (RRF)

Combining Multiple Search Results

When you run multiple query variations (like "How does Tesla make money?", "What are Tesla's revenue sources?", "Tesla business model profitability"), you get separate ranked lists of chunks.

Now this begs the question, how do you combine them into one final ranking?

Simply concatenating them isn't ideal because:

- The same chunk might appear in multiple lists at different positions
- Some queries might return more relevant results than others
- You want to boost chunks that appear in multiple result sets

And this is where Reciprocal Rank Fusion comes into the picture.

Reciprocal Rank Fusion (RRF)

How RRF solves this problem:

Reciprocal Rank Fusion (RRF) is a method that combines multiple ranked chunk lists by giving each chunk a score based on its positions across all lists.

RRF_score = Σ (1 / (k + rank_position))

Where:

- k is a constant (typically 60)
- rank_position is the position of the chunk in query result
- The sum is across all queries where the chunks appears

Reciprocal Rank Fusion (RRF)

Retrieval A

Query: What are the revenue streams for Tesla?

1.	Chunk X	1/1 = 1
2.	Chunk Y	½ = 0.5
3.	Chunk Z	⅓ = 0.33
4.	Chunk W	¼ = 0.25
5.	Chunk V	⅕ = 0.20

Retrieval B

Query: In what ways does Tesla generate income?

1.	Chunk Y	1/1 = 1
2.	Chunk X	½ = 0.5
3.	Chunk A	⅓ = 0.33
4.	Chunk B	¼ = 0.25
5.	Chunk Z	⅕ = 0.20

RRF Calculation (Simple Version)

Chunks appearing in both retrievals:

Chunk X:

- Position 1 in Retrieval A: 1/(0+1) = 1.0
- Position 2 in Retrieval B: 1/(0+2) = 0.5
- Total RRF Score: 1.0 + 0.5 = 1.5

Chunk Y:

- Total RRF Score: 0.5 + 1.0 = 1.5

Chunk Z:

- Total RRF Score: 0.33 + 0.2 = 0.53

Chunk W (only in A): 1/4 = 0.25
Chunk V (only in A): 1/5 = 0.2
Chunk A (only in B): 1/3 = 0.33
Chunk B (only in B): 1/4 = 0.25

* We need to send unique 5 highest ranking chunks for the final answer gen

RRF_score = Σ (1 / (k + rank_position))

Reciprocal Rank Fusion (RRF)

Final Ranking:

Chunk X & Chunk Y: 1.5 (tie! - both appear highly in both queries)

Chunk Z: 0.53

Chunk A: 0.33

Chunk W & Chunk B: 0.25 (tie)

Chunk V: 0.2

Reciprocal Rank Fusion (RRF)

The Problem with k=0

Notice how dramatic the score differences are:

- Position 1 gets score 1.0
- Position 2 gets score 0.5 (50% penalty!)
- Position 3 gets score 0.33 (67% penalty!)

Real-World Issue:

In our example, Chunk X and Chunk Y might have had very similar similarity scores (0.95 vs 0.94 in Retrieval A).

That tiny 0.01 difference in similarity gets amplified into a 2x scoring difference (1.0 vs 0.5) in RRF!

This means we're treating a 1% similarity difference as if one document is twice as good as the other. That's problematic because:

- We're over-penalizing lower positions
- Small differences in similarity scores often don't reflect meaningful relevance differences

Reciprocal Rank Fusion (RRF)

Query variation 1 Retrieval Results

1. Chunk X — 1/(60 + 1) = 0.0164
2. Chunk Y — 1/(60 + 2) = 0.0161
3. Chunk Z — 1/(60 + 3) = 0.0159
4. Chunk W — 1/(60 + 4) = 0.0156
5. Chunk V — 1/(60 + 5) = 0.0154

Query variation 2 Retrieval Results

1. Chunk Y — 1/(60 + 1) = 0.0164
2. Chunk X — 1/(60 + 2) = 0.0161
3. Chunk A — 1/(60 + 3) = 0.0159
4. Chunk B — 1/(60 + 4) = 0.0156
5. Chunk Z — 1/(60 + 5) = 0.0154

Final Ranking with k=60:

1. Doc X & Doc Y: 0.0325 (still tied, but for good reason - true consensus!)
2. Doc Z: 0.0313
3. Doc A: 0.0159
4. Doc W & Doc B: 0.0156
5. Doc V: 0.0154

RRF_score = Σ (1 / (k + rank_position))
k = 60 (constant)

Since the differences between positional scores is significantly less, even the lower ranking chunks have a shot at having higher RRF scores if they appear in multiple retrievals

Reciprocal Rank Fusion (RRF)

Key takeaways:

1. RRF boosts chunks that appear in multiple query results - this is the "consensus" effect
2. k=0 is too harsh - it over-emphasizes top positions and under-values lower positions
3. k=60 provides balance - top positions still matter, but the differences are more reasonable (widely used in popular projects)
4. RRF preserves diversity - unique chunks from each query still contribute to the final ranking
5. Simple but effective - despite its simplicity, RRF often outperforms more complex fusion methods

Hybrid Search

Problem with the pure vector search:

- Vector search is great at understanding semantic meaning and context
- But it has some blind spots that can hurt retrieval quality
- It struggles with exact matches - like user searching for "API key authentication" but getting results about "user login security"
- It can miss important keywords that users specifically mention
- Sometimes users know exactly what terms they want to find

This is where Keyword Search comes in (The Classic Approach):

- Keyword search looks for exact word matches in your chunks
- It's been around forever (think Google in the early 2000s)
- Super reliable for finding specific terms, names, model numbers, technical jargon
- But it's "dumb" - it doesn't understand that "car" and "automobile" mean the same thing

Hybrid Search

Hybrid Search

The Algorithm Behind Keyword Search: BM25

BM25 (Best Matching 25) is the gold standard for keyword search

It scores chunks based on two key factors:

- **Term Frequency (TF)**: How often does the search term appear in this specific chunk?
- **Inverse Document Frequency (IDF):** How rare is this term across your entire chunk collection?

Example:

TF Example: Chunk A says "Tesla's revenue growth is strong. Tesla reported record profits." vs Chunk B says "Tesla reported strong results." → "Tesla" query gets us Chunk A because it mentions "Tesla" twice

IDF Example: If "Tesla" appears in 100 chunks but "Cybertruck" appears in only 5 chunks → For the query "Tesla Cybertruck", those 5 chunks gets higher IDF score because it's rarer

Combined: For query "Tesla Cybertruck", a chunk with "Tesla's Cybertruck production" beats a chunk with "Tesla Tesla Tesla financial reports" because the rare term "Cybertruck" is more valuable than repeated common terms

BM25 is an improved version of TF-IDF that prevents keyword stuffing and handles chunk length better

Hybrid Search

Why We Need Both: The Hybrid Approach

- Vector search catches semantic relationships and context
- Keyword search catches exact matches and specific terminology
- Together, they cover each other's weaknesses
- Think of it as having both a smart assistant (vectors) and a precise librarian (keywords) working together

Ensemble Retriever RRF With Weights

Vector Search Retrieval

1.	Chunk X	0.7/(60 + 1) = 0.0115
2.	Chunk Y	0.7/(60 + 2) = 0.0113
3.	Chunk Z	0.7/(60 + 3) = 0.0111
4.	Chunk W	0.7/(60 + 4) = 0.0109
5.	Chunk V	0.7/(60 + 5) = 0.0108

Keyword Search Retrieval

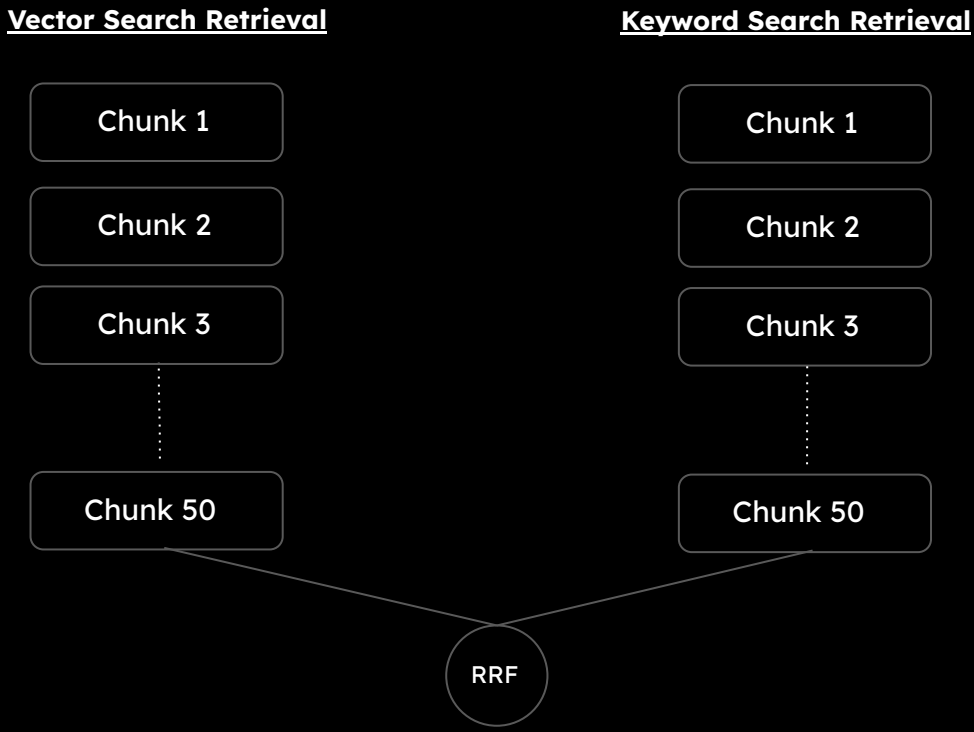
1.	Chunk Y	0.3/(60 + 1) = 0.0049
2.	Chunk X	0.3/(60 + 2) = 0.0048
3.	Chunk A	0.3/(60 + 3) = 0.0048
4.	Chunk B	0.3/(60 + 4) = 0.0047
5.	Chunk Z	0.3/(60 + 5) = 0.0046

Final Ranking with k=60:

1. Doc X: 0.0115 + 0.0048 = 0.0163
2. Doc Y: 0.0162
3. Doc Z: 0.0157
4. Chunk W: 0.0109 + 0.0000 = 0.0109 (not in BM25)
5. Chunk V: 0.0108 + 0.0000 = 0.0108 (not in BM25)
6. Chunk A: 0.0000 + 0.0048 = 0.0048 (not in Vector)
7. Chunk B: 0.0000 + 0.0047 = 0.0047 (not in Vector)

RRF_score = Σ (1 / (k + rank_position))
k = 60 (constant)

Reality



Result: ~30 chunks

Reranking


1. User Query
"What is the capital of France?"
2. Multiple Retrievers
Dense, Sparse, Hybrid Retrieval Models
3. Individual Rankings
Each retriever ranks documents
4. RRF Fusion
Combine rankings using RRF
RRF(d) = Σ 1 / (k + r(d))
5. Final Ranking
Unified document ranking
6. Generation
Generate answer using top docs
Legend:
d: document being ranked
k: constant (typically 60)
r(d): rank of document d
Σ: sum across all rank lists

Reranker

What is it?

A reranker is a specialized AI model that acts as a "quality inspector" that improves the initial search results by reordering them based on semantic relevance to your query.

Think of it as a "second opinion" system. Your initial retrieval (result of hybrid + RRF) might return 100 chunks.

The reranker takes these chunks and applies more sophisticated analysis to reorder them by putting the most relevant chunks at the top.

Reranker

Why do we even need a reranker?

"I already have vector embeddings, hybrid search, and at the end, RRF (Reciprocal Rank Fusion) combining my results. I can extract the top K chunks from that set alone. Why do I need another layer of complexity with rerankers? Isn't my hybrid + RRF results already good enough?"

Embeddings Limitation:

Query: "How to fix a leaky faucet?"

Chunk A: "To repair a dripping tap, first turn off the water supply, then remove the handle and replace the worn washer."

Chunk B: "Water damage from leaky faucets can cost homeowners thousands in repairs and lead to mold growth."

Embedding similarity scores:

Chunk A: 0.82 (high similarity)

Chunk B: 0.79 (also high similarity!)

Although embeddings and similarity scores help us retrieve chunks that have a good probability of being relevant, "good" is not good enough when you need a HIGH-accuracy RAG system.

Reranker

The Two-Stage Strategy:

Stage 1: Embeddings (Fast & Broad)

Query → Vector → Compare with 1M+ chunks → Top 100 chunks

Purpose: Inexpensive approximation

- Fast: Can search millions of chunks quickly
- Broad: Good at finding chunks in the right neighborhood
- Cheap: Low computational cost per comparison
- Reliable: Gives you chunks with good probability of relevance

BUT: It's just an approximation. You wouldn't bet your money on the exact ranking.

Wouldn't it be great if there was a AI model that isn't as expensive as an LLM model but more accurate than (simple embeddings + similarity scores) that could reorder the final 20-30 chunks accurately based on user query?

Reranker

The Two-Stage Strategy:

Stage 2: Reranker (Precise & Focused)

Query + Each of 25 chunks → Reranker/Cross-encoder → Precise relevance scores

Purpose: Increase probability that top 10 are the absolute best

- **Precise:** Actually reads query and chunk together
- **Context-aware:** Understands query intent and relationships
- **Expensive:** Higher computational cost, but only for 10-100 chunks usually

Why This Two-Stage Approach Works:

#1 Embeddings: Cast a wide, inexpensive net to find chunk candidates

#2 Reranker: Apply slightly more expensive but precise analysis to finalize ranking

It's like having a screening interview followed by a detailed technical interview - each stage optimized for its purpose.

Reranker

How does reranker (cross-encoder) achieve this?:

Embeddings (Bi-encoder approach):

```
Query: "apple stock price" → [0.1, 0.8, 0.2, ...] (vector)
Chunk: "Apple trees grow in orchards" → [0.3, 0.7, 0.1, ...] (vector)
Similarity = cosine(query_vector, chunk_vector) = 0.78
```

Problem: The model never sees the query and chunk together. It processes them separately, then we just compare the math.

Reranker

How does reranker (cross-encoder) achieve this?:

Reranker (Cross-encoder/joint-processing approach):

```
Combined Input: "apple stock price [SEP] Apple trees grow in orchards"
    ↓ (processed together)
Cross-encoder analyzes relationship
    ↓
Relevance Score: 0.12 (correctly identifies mismatch)
```

Advantage: The reranker model reads both the query and chunk simultaneously, understanding their relationship.

Reranker

Conclusion:

The Trade-off: Cross-encoders are more computationally expensive because they need to process each query-chunk pair individually. But since we're only reranking 20-100 chunks (not millions), this cost is less for the accuracy gain.

Your multi-query hybrid search + RRF gets you to "good probability" candidate chunks.

The reranker gets you from "good probability" to "highest confidence" ranking for the final results that actually reach your LLM.